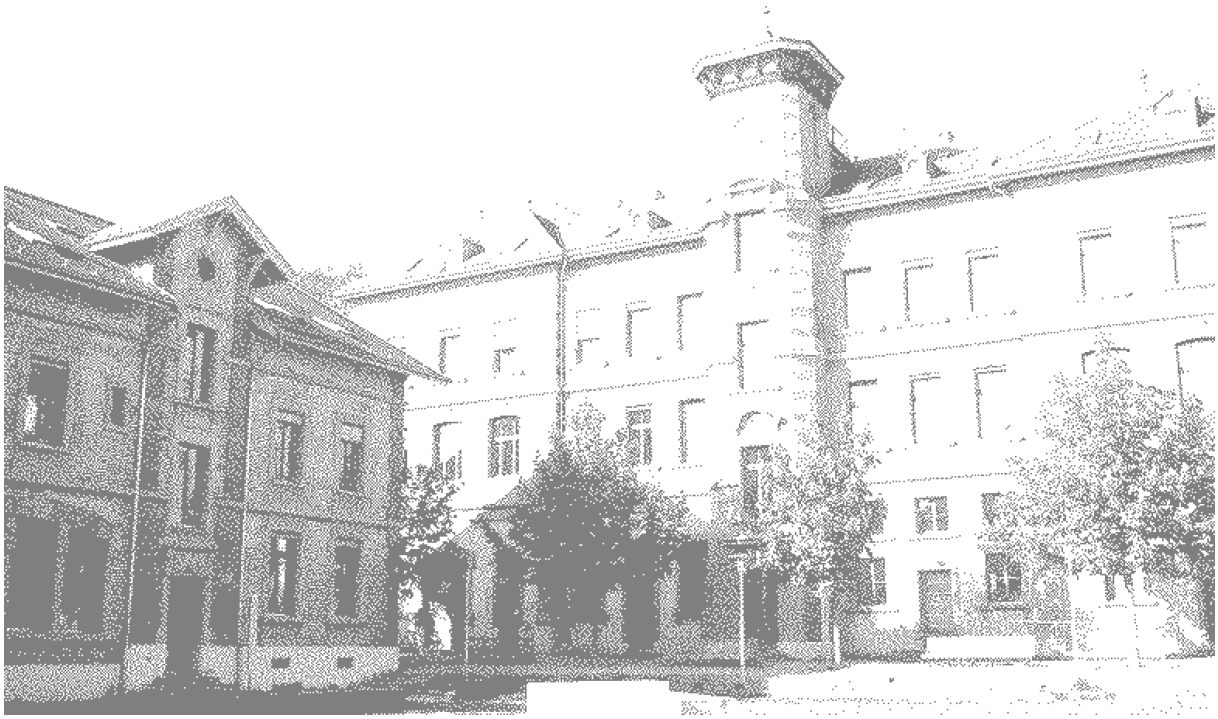


PyGirl
Generating Whole-System VMs from high-level models using PyPy

C. Bruni, T. Verwaest, M. Denker

Technischer Bericht IAM-09-002 vom 21. Januar 2009

Institut für Informatik und angewandte Mathematik, www.iam.unibe.ch

PyGirl
Generating Whole-System VMs from high-level models using PyPy

Camillo Bruni, Toon Verwaest, Marcus Denker

Technischer Bericht IAM-09-002 vom 21. Januar 2009

CR Categories and Subject Descriptors:
D.1.5 Object-oriented Programming; C.4 [Performance of Systems]: Design studies

General Terms:
Languages, Performance

Additional Key Words:
pypy, python, rpython, whole system virtual machine, translation toolchain, high-level language

Institut für Informatik und angewandte Mathematik, Universität Bern

Abstract

Virtual machines emulating hardware devices are generally implemented in low-level languages and using a low-level style for performance reasons. This trend results in largely difficult to understand, difficult to extend and unmaintainable systems. As new general techniques for virtual machines arise, it gets harder to incorporate or test these techniques because of early design and optimization decisions. In this paper we show how such decisions can be postponed to later phases by separating virtual machine implementation issues from the high-level machine-specific model. We construct compact models of whole-system VMs in a high-level language, which exclude all low-level implementation details. We use the pluggable translation toolchain PyPy to translate those models to executables. During the translation process, the toolchain reintroduces the VM implementation and optimization details for specific target platforms. As a case study we implement an executable model of a hardware gaming device. We show that our approach to VM building increases understandability, maintainability and extendability while preserving performance.

Contents

1 Introduction

The research field revolving around virtual machines (VMs) is mainly split up in two big subfields. On the one hand we have the whole-system VM (WSVM) community working on new ways of building and optimizing emulators for hardware devices. These VMs have close resemblance to how the actual hardware which they are emulating. On the other we have the language community building high-level language VMs (HLLVM). These VMs *only* exist virtually. There are no hardware counterparts which natively understand the code running on those VMs. Although both domains share conceptual and implementation similarities, only recently the awareness has been growing about the overlap of ideas and acknowledgement that both fields can learn from each other. As a clear example of this fact we see that modern VM books discuss both fields [1]. But while up until now almost only performance enhancement techniques have been adapted, we are confident that more can be shared.
In this paper we show how the effort of separating language specific VM models from the general implementation and optimization details, which has been developed for HLLVMs in the PyPy project[1] can also greatly improve understandability, maintainability, extendability and possibly even performance optimization of WSVMs. Similar to Spy [2] we will build a custom high-level virtual machine model, but this time rather than implementing a HLLVM, we will concentrate on emulating a hardware gaming device. We focus on building a high-level, abstract but executable model rather than an inflexible, early optimized system. We will then show that by using the PyPy toolchain, we are able to generate performant low-level virtual machines from those models.
The main contributions of this paper are

- We show how the execution and implementation details of WSVMs can separated in the same way as this was previously done for HLLVMs
- We provide an implementation of a WSVM model for PyPy, for which we show the improved readability, maintainability and understandability without loss of performance

The paper is structured as follows. In Section 2 on page 3 we will give an introduction to the PyPy project. Section 3 on page 7 covers the technical details of the emulated game device, followed by the actual implementation

[1] `http://codespeak.net/pypy/`

details of PyGIRL in Section 4 on page 9. In Section 5 on page 18 we compare the performance of the different WSVMs implementations. We provide an outlook of our future work in Section 6 on page 21. Then finally we provide a brief overview of our achievements in Section 7 on page 23.

2 PyPy in a Nutshell

In this section we describe the PyPy project, which we use as a translation toolchain.
The goal of PyPy was to write an full featured, customizable and fast interpreter for Python written in Python itself, in order to have the language described in itself. Of course, running an interpreter on top of another interpreter results in slow execution, and still requires a first interpreter written in another language below it. For this reason it was decided to build a "domain specific compiler", a toolchain which translates high-level specifications of interpreters (VMs) in Python down to low-level backends, such as C/Posix [3]. Like the interpreter, the toolchain itself is written in Python. This effort is similar to other self-sustaining systems like Squeak where the VM is written in Slang [4]. The major difference between Slang and PyPy is that Slang is a thinly veiled Smalltalk-syntax on top of the semantics of C, whereas PyPy focuses on making a more complete subset of the Python language translatable to C. This difference is clearly visible in the level of abstraction used by programs written for the respective plattforms.

2.1 The Interpreter

The starting point for PyPy was to create a minimal but full interpreter for Python written in Python itself. With minimal we mean that all interpreter implementation details such as garbage collector and optimizations are not implemented. For these features the PyPy interpreter rather relies on the garbage collector and optimizations available in the environment running it. This results in a very clean and concise implementation of the interpreter which models how the language works without obscuring implementation details.

2.2 The Translation Toolchain

Since high-level executable models of VMs will not run fast by themselves, PyPy also implements a "domain specific compiler", which can translate VMs in Python down to low-level code. This translator is designed as a flexible toolchain where front- and backends can be replaced so that it can generate VMs for different languages which will run on different platforms.

Not only the front- and backend can be changed, but also different aspects of the translation itself. This results in fast and portable VMs.

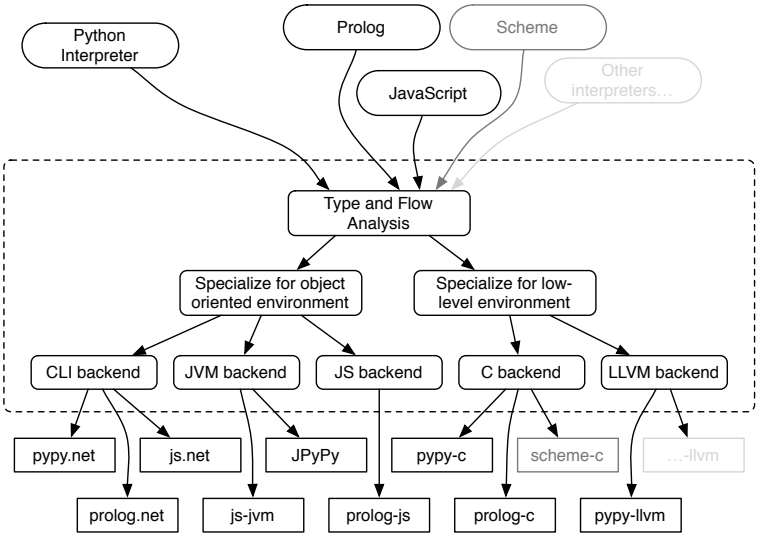


Figure 1: PyPy translation toolchain architecture

First the toolchain builds a modifiable and dynamic flow graph from the target program's sources. From there on, this model is transformed in several steps until the final binary results. Using the same intermediate representation for a large part of the translation and applying transformations in small steps allows us to add custom modifications for every translation aspect.

2.2.1 Translation Steps

In the first step the translator loads the source-code it will translate. Unlike standard compilers which would start by parsing the source code, the PyPy toolchain never even sees the Python source code. Since the input code for the translator is RPython code (which is a subset of Python code), and the toolchain itself is running on top of a Python interpreter, the toolchain can use its hosting Python interpreter to load its input files. While loading, the Python interpreter will evaluate all top-level statements and add the loaded method and class definitions to the global Python memory. After loading, the toolchain uses the the globally loaded `main` function as starting point for its input graph. This setup allows the input model to apply metaprogramming in plain Python code at preprocessing time, as long as

the resulting graph in memory is RPython compatible (Listing 2 on page 12 shows an example of this feature). From the code objects resulting from loading the sources, a control flow graph is generated.
Because PyPy mostly targets statically typed backends, the graph is annotated with inferred types. Starting with a specified entry point, the type inference engine works its way through the object flow graph and tries to infer most specific types. If multiple types are possible for a certain node, the type inference engine will try to select the common superclass as type. If the most specific type found for a node would be the most common type object type in the system, an error is thrown to show that no specialization was possible. The same happens when two types are incompatible, like booleans and objects. If such type-errors arise, they have to be solved by the programmer. Such problems are often solved by introducing a new common superclass or moving a method higher up the hierarchy. In other cases they really are semantical errors and require restructuring (Section 4.3 on page 14 covers some aspects of resolving errors discovered by the toolchain). Here we see that the compiling process has an impact on the structure of the input source code. It effectively limits the expressiveness of the input language. For this reason we call the input language understood by the PyPy toolchain RPython instead of Python, as we will see in Section 2.3 on the following page.
The annotation step is followed by the conversion from a high-level flow-graph into a low-level one. There is currently a converter which specializes towards low-level backends and one which specializes towards object-oriented backends (Figure 1 on the preceding page).
On the low-level flow-graph, optional backend optimizations are performed. These optimizations are rather similar to optimizations found in standard compilers, like function inlining and escape analysis.
After the low-level flow-graph is optimized, it gets specialized for a specific backend. The preparation for code generation covers the following points:

- insertion of explicit exception handling
- adding memory management details. Different garbage-collection strategies are available[2]. Note that these garbage collectors themselves are also written in Python code. They also get translated and woven into the VM definition.
- creation of low-level names for generated function and variables

[2] `http://codespeak.net/pypy/dist/pypy/doc/garbage_collection.html`

Finally the language-specific flow-graph is transformed into source files. These source files are then again compiled or assembled to binaries by a language-specific compiler or assembler, which can perform further domain-specific optimizations. For example generated C source files are compiled with GCC with the `-O3` flag.

2.3 RPython

In order to boost the performance of the Python interpreter written in Python, the PyPy translation toolchain was created. The translation from a dynamically typed language like Python to a statically typed language like C is however not straightforward. As mentioned before, in order to do this in a semantically correct way, we are enforced to limit the expressiveness of the input language. For this reason, when we talk about the language understood by the translation toolchain, we do not call this Python but rather RPython or restricted Python. The language is rather imprecisely defined as Python with the following restrictions applied:

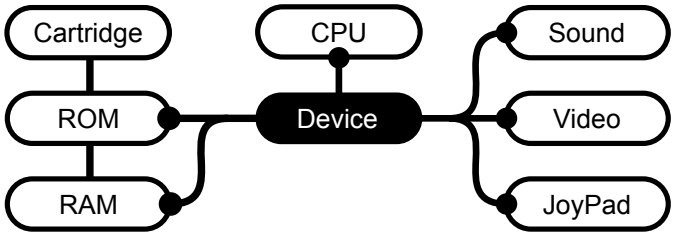
- Variables need to be type consistent
- Runtime reflection is not supported
- All globals are assumed to be constants
- All code has to be type inferable

The complete definition of RPython itself is fuzzy as it is defined by the evolution of the translator and the community.[3]
Although these restrictions seem to be substantial for a dynamic language such as Python, it is still possible to use high-level features like single inheritance, mixins and exception handling. More importantly, since RPython is a proper subset of Python, it is possible to test and debug the input program with the known Python tools before trying to translate it using PyPy. As such the VM models we build for PyPy are executable by themselves, thus giving a great speedup against a classical compile-wait-test cycle.

[3] `http://codespeak.net/pypy/dist/pypy/doc/coding-guide.html#restricted-python`

3 Gaming Device Hardware Technical Details

In this section we list the technical details of the gaming hardware. An official documentation is available on the producer's website.[4]



3.1 Hardware Pieces

The system is composed of 6 essential pieces which are accessible through shared memory. External events are supported through a 8 bit maskable interrupt channel. The use of an 8 bit processor makes the handling of opcodes compact and maintainable. There are two kind of opcodes:

- First order opcodes are executed directly
- Second order opcodes fetch the next instruction for execution. The combined opcode doubles the range of possible instructions at the cost of execution speed. The second order opcodes are mostly used for bit testing and bit setting on the different registers.

The following list show some more specific details of the different parts of our gaming device.

- The 8 bit CPU is a slightly modified version of the Zilog 80 with a speed of 4.19 MHz.

  The CPU supports two power-saving mechanisms, which both work in a similar way. After a certain interrupt, the CPU is put into a low power consumption mode which is left only after another interrupt occurred.

[4] `http://www.nintendo.co.uk/NOE/en_GB/support/game_boy__pocket__color_559_562.html`

- The cartridge contains ROM with the embedded game and possibly additional RAM and/or other devices. The size of the RAM is depending on the type of cartridge. Some types of cartridges support an additional battery to store game-state. Switchable memory banks are used to extend the 8 bit limited address range. A checksum in the header and a startup procedure are used to guarantee that the device is working correctly and the cartridge is not corrupted.

- The supported Resolution is 160×144 Pixels with 4 shades. It is possible to show maximally 40 Sprites of 8×16 or 16×16 pixels at the same time. The video chip has two tile-map registers, one for the background and one for the foreground. A masking window can be used to crop the background thus enabling basic low-level support for scrolling.

- A serial connection can be used to communicate with another device. This is useful for multiplayer games.

- The sound chip supports stereo sound and has four internal mono sound channels. Sound can be either read directly from the RAM, thus creating arbitrary sound at the cost of its calculation, or it can be produced via a noise-channel or via two different wave pattern generators.

4 PyGirl Implementation

In this section we will highlight some specific implementation details of the PyGirl VM.
For our implementation we did not start from a formal specification of the hardware. Instead we chose to adapt an existing stereotypical implementation of a VM for the same device to PyPy style for easier comparison.

4.1 Source Implementation

The VM implementation we chose as a basis [5] is a VM written in Java. It is developed in a portable manner. By abstracting out platform-specific details to certain components, they are able to provide variants of the emulator for the different versions of Java architectures like the Java Standard Edition and Applets for the web. The application is structured by providing mappings for each piece of hardware to one class. Platform-specific parts are factored out by providing a set of abstract driver interfaces which handle input and output. These drivers are then implemented for each architecture separately, adapting to the requirements. Even though this is a fairly abstract and portable design, this again shows that by implementing your VM without a toolchain, your VM codebase will be cluttered with backend specific details.
While the code is written in an object-oriented manner at first glance, many parts of the Java system strictly follow the low-level execution details of the hardware. On top of this, the implementation is cluttered with local optimizations. Two types of optimization strategies clearly stand out: manual inlining of code and manual unrolling of loops. Both strategies result in an overly expanded code-base, obscuring the overall design and semantics.
For example, the CPU class is cluttered with such speed optimizations. The reason is that a CPU is a very low-level general-purpose device which provides less possibilities for abstraction than others. However, even the video chip is implemented in a non-abstract procedural way. This even though there are more conceptual components ready for abstract representation, such as *sprites*, *background* and *foreground*.

4.2 From Java to Python

We started off by migrating the source VM from Java to Python, without applying any optimizations or refactorings. Like this we could easily track

the progress we made while refactoring. During the whole process, we kept the overall structure of the existing system as it directly corresponds to the hardware.
The following sections cover different refactorings we applied and abstractions we introduced, to go from a low-level detailed implementation to a high-level model of the VM.

4.2.1 Memory Usage Considerations

The Java code is cluttered with type-casts between bytes and integers. Bytes are used to represent the 8 bit hardware architecture, whereas the integers are used for all kind of operations. Instead of using integers whenever possible, the Java version focuses on reducing the memory footprint of the running emulator and even more, it focuses on the implementation details of the original hardware. Only at very few places in the code was the usage of bytes justified by the resulting two's-complement interpretation of the numbers.
In our model type-casts are removed wherever possible to improve readability. Firstly we consider the memory footprint of the device we will emulate too small to justify optimizing memory usage. Even if we use four to eight times as much memory as the original device would have used, corresponding to an expansion from 8 to 32 or 64 Bit, this means that we will end up with about 20Mb memory usage, which is a negligible amount for modern computers. Secondly and more importantly, it is (hypothetically speaking) possible to plug an additional transformation into the toolchain, which converts all integers to bytes, so that the memory footprint of the final VM is equal to the original device.

4.2.2 The God Switch

The most prominent candidate for refactoring duplicated and inlined code is the CPU class. In the original version the class is around 4000 lines long, featuring an unpleasant 1700 line switch which delegates the incoming opcodes.

```
public void execute(int opcode) {
    switch (opcode) {
    case 0x00:
        this.nop();
        break;
```

```
    ...

    case 0xFF:
        this.rst(0x38);
        break;

    default:
        throw new RuntimeException(ERR);
    }
}
```

On top of that, there is a nested switch of 800 LOC, which handles the second order opcodes. In both switches we could identify patterns which could be used as basis for extra abstractions. In the following excerpt from the original Java code we can see how bytecodes directly encode their semantics in a structured way

```
public void execute(int opcode) {
    ...
    case 0x78:
        this.ld_A_B();
    case 0x79:
        this.ld_A_C();
    ...
}

public final void ld_A_B() {
    this.a = this.b;
    this.cycles -= 1;
}
```

Listing 1: Java: Grouped opcode mappings

The Java code covers all these cases by manually specifying them in the switch and by encoding the logic in one function per opcode. Since all these operations are symmetrical in terms of semantics and usage of cycles, even while there are only few lines of code per operation, there is quite some redundancy. The opcodes in Listing 1 encode the arguments which should be passed to the `load` function. Instead of separated functions PyGirl uses a `load` for different registers such that it can be reused for more than one opcode.

```
def load(self, register1, register2):
   register1.set(register2.get())
```

To refactor Listing 1 on the previous page, we created such reusable functions for all the different types of operations. Then we simply replaced the whole switch with a compact lookup in an opcode table which we generate from the abstract functionality descriptions using metaprogramming.

```
def execute(self, op_code):
    OP_CODES[op_code](self)
```

Instead of hardcoding the mapping to the respective functions, we used metaprogramming to compute the definition at translation time. The mapping of opcodes to function in the example Listing 1 on the preceding page can be replaced with the following flexible definition.

```
def create_op_codes(table):
    op_codes = []

    for entry in table:
        op_code = entry[0]
        step = entry[1]
        function = entry[2]

        for getter in entry[3]:
            op_codes.append(
                (op_code,
                 register_lambda(function,
                                 getter) ))
            op_code += step

    return op_codes


def register_lambda(function, registerOrGetter):
    if callable(registerOrGetter):
        return lambda s: function(s,
                       registerOrGetter(s))
    else:
        return lambda s: function(s,
                       registerOrGetter)


REGS = [CPU.get_bc, ..., CPU.get_sp]

SET = [
```

```
    (0x01, 0x10, CPU.fetch_double_register, REGS),
    (0x03, 0x10, CPU.inc_double_register, REGS),
    (0x09, 0x10, CPU.add_hl, REGS),
    (0x0B, 0x10, CPU.dec_double_register, REGS),

    ...
    (start, step, func, REGS)
    ...
]

OP_CODE_TABLE += create_op_codes(SET)
```

Listing 2: RPython: Opcode generation at preprocessing time using metaprogramming

The `create_op_codes`-function creates a part of the final opcode table from a list of inputs. Each input entry consists of a start opcode, the offset to add to get the following opcode and finally an ordered list of registers. The input set allowed us to keep a clean and compact notation for the variety of opcode mappings.
We used the static creation of opcode table entries for most of the register operations, such as loading and storing, but also for nearly all other register operations. In total we were able use metaprogramming for about 450 out of all 512 opcodes.
As a note on performance, when translating this code to C, PyPy is able to take the preprocessed opcode table and translate it back into an optimized switch. So the source code stays compact and maintainable, without loss in performance. Even better, PyPy should be able to optimize the switch at runtime so that often used branches are scheduled first. PyPy can then also inline small methods. Although the author of the Java version didn't inline any method, it would have been the next logical step to optimize the code in a classical way. By letting PyPy handle the optimizations, we maintain a clean implementation.

4.2.3 Abstraction

Since the Java implementation tries to evade methods calls, different internal subparts of the hardware device were implemented as simple byte or integer fields rather than separate high-level objects. This was mostly likely again for memory footprint reasons.
Since we care more about having a nicely designed high-level model of the VM, we introduced classes for the different registers and the interrupt

channel. This new abstraction reduced the usage of byte operations in many places. As another abstraction, we created a flag register to handle the different states and results of CPU operations. In the original code every single bit of the flag had to be extracted with a mask and a had to be checked with an integer or byte comparison, cluttering the code once more. Our flag register consists of 8 boolean fields which can be accessed by a representative name, hence increasing the understandability and maintainability of the code.
The following example shows the code of two flag-related methods in Java and the corresponding ones in Python. The `ccf`-method inverts the carry-flag and resets all the other flags but the zero-flag. The `scf`-method resets all flags except for the zero-flag and sets the carry-flag:

```
public final void ccf() {
    this.f = (this.f & (Z_FLAG | C_FLAG)) ^ C_FLAG;
}

public final void scf() {
    this.f = (this.f & Z_FLAG) | C_FLAG;
}
```

The corresponding code in PyGirl is not that compact, but more importantly the understandability is increased:

```
def complement_carry_flag(self):
    self.flag.partial_reset(keep_is_zero=True,
                            keep_is_carry=True)
    self.flag.is_carry = not self.flag.is_carry


def set_carry_flag(self):
    self.flag.partial_reset(keep_is_zero=True)
    self.flag.is_carry = True
```

4.3 Translation

We now describe some of the problems we encountered while using PyPy to translate our VM down to a low-level backend. Such problems typically arise because of differences between Python and its restricted subset, RPython, described in Section 2.3 on page 6. Generally these problems only require small refactorings and only come up when the VM is already in a pretty stable state. This results from the fact that our executable model

fully runs on a normal Python interpreter, which allows us to execute first prototypes without having to care if it can already be translated with PyPy. We show some examples and how to solve these specific cases so that the translation can succeed.
The easiest translation bugs are related to straightforward syntactic bugs, like typos. After these inconsistencies are cleared out, we encountered typical type conversion errors. Some of those problems were related to confusions between integers and floats, others were related to wrong inheritance. For some classes we introduced abstract superclasses so that they would have a common type.

4.3.1 Call Wrappers

We handled operations on registers and other CPU functions elegantly by just passing around the function closures, allowing us to reuse methods for different actions. A very common example is the following `load` method:

```
def load(self, getter, setter):
    setter(getter())

...

load(self.flag.get, self.a.set)
load(self.fetch, self.a.set)
```

`load` is called with different arguments. In the first example it is used to copy the values from the flag-register into the register `a`. The second example loads the next instruction into register `a`. Due to the different origins of the passed function closures, the toolchain was unable to transform this into a typed counterpart. As solution we created call-wrappers with a common superclass. Introducing call-wrappers for the passed function closures added some overhead, but it was a simple way to keep the code minimal and close to the original idea of passing around closures. The following code shows the same function calls using wrappers for each passed type of closure

```
class CallWrapper(object):
    def get(self, use_cycles=True):
        raise Exception("called CallWrapper.get")

    def set(self, value, use_cycles=True):
        raise Exception("called CallWrapper.set")
```

```
class RegisterCallWrapper(CallWrapper):
    def __init__(self, register):
        self.register = register

    def get(self, use_cycles=True):
        return self.register.get(use_cycles)

    def set(self, value, use_cycles=True):
        return self.register.set(value, use_cycles)


class CPUFetchCaller(CallWrapper):
    def __init__(self, CPU):
        self.CPU = CPU

    def get(self, use_cycles=True):
        return self.CPU.fetch(use_cycles)
```

`RegisterCallWrapper` takes a register and calls `get` or `set` on it. The `CPUFetchCaller` is used as an abstraction for the `fetch` method of the CPU so that it can be used as a possible argument for the `load`-method. These are two out of the 5 total call-wrappers we used to handle closure passing. The common superclass `CallWrapper` makes it possible for PyPy to infer a common type for the argument of every method. Something which is not that clearly visible in this code is the fact that every `get`-method of the call-wrapper returns an integer. Thus all methods would have the same signature. To support the call-wrappers we replaced the closure calls by a `get` or `set` on the call wrappers. Applied to the `load` method this resulted in the following code

```
def load(self, getCaller, setCaller):
        setCaller.set(getCaller.get())
```

Then the method-calls from the original example look now like

```
load(RegisterCallWrapper(self.flag),
    RegisterCallWrapper(self.a))

load(CPUFetchCaller(self),
    RegisterCallWrapper(self.a))
```

4.4 State of the Implementation

The current version supports simple test ROMs without sound and moving sprites. Static images and bit-blitting are supported. This means that most games are able to show an intro image but there is no animation. In the current version, the video part is heavily redesigned, using high-level definitions for all the logical parts in the video chip. Because we mostly focussed on the graphics part, the sound chip has not yet been implemented for PyGirl. The CPU is fully functional and like all other parts, completely test-covered.

5 Performance evaluation

We ran the emulation on three different flavours of the VM. The original Java emulation, the interpreted variant of our implementation and finally the translated version. The interpreted version was run on top of the Python 2.5 Interpreter[5], whereas the translated version was built from those sources.
The benchmarks are shown for the original Java implementation, the interpreted sources of PyGirl and finally for the translated binaries of PyGirl. Each test shows the average execution time over 1000 runs using Java `1.6.0_06`, `1.5.0_15` and Python `2.5.2` on a 64 Bit Ubuntu `8.04.1` server machine with an Intel Xeon CPU QuadCore 2.00 GHz processor. We use revision number `59328` of the PyPy project. PyPy uses the following GCC optimizations when creating the binary executable:

```
-O3 -fomit-frame-pointer -pthread -I/usr/include/python2.5
```

The complete GCC calls and arguments are visible during the end of the translation process. Instructions for running the benchmarks are given in Appendix 11 on page 26.

5.1 All opcodes

This benchmark covered all possible opcodes which were executed equally distributed, giving a theoretical speed of the WSVM. The results of this benchmarks are shown in Figure 2 on the next page.
It is clearly visible that the interpreted version running on top of CPython is approximately 100 times slower than the Java version and 250 times slower than the translated version. This test compares the overall performance of the system, which is no directly related to the performance of the emulator running a game.

5.2 Typical opcode Set

To compare the performance of the emulator in a typical situation, we considered in this test a weighted opcode set which we constructed by running multiple games. From these test-runs, the 30 most executed opcodes are selected, as they probably represent typical in-game performance. The

[5] `http://python.org/download/`

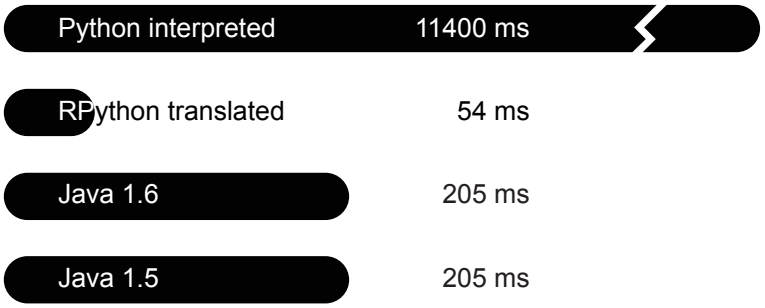


Figure 2: Benchmark: Comparing the execution time of PyGirl with the Java version weighting all opcodes equally

selected opcodes are listed in Table 2 on page 28. The results of this benchmarks are shown in Figure 3

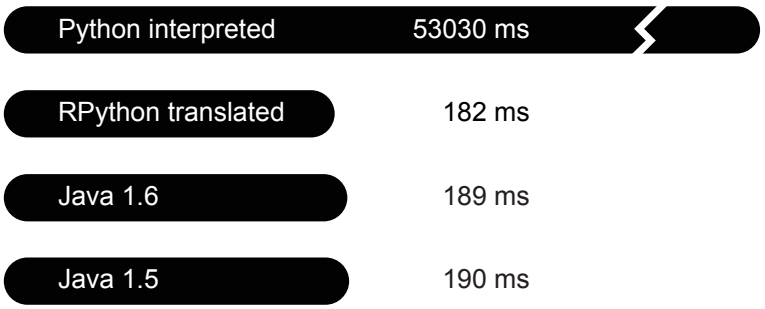


Figure 3: Comparing the execution time of PyGirl with the Java version weighting the opcodes according to a typical emulation run

The results differ from the first benchmark, which has its origins in the different relative execution times of opcodes in the Java and in the translated version. Java also was able to optimize the code at runtime, using a JIT. While this will also be available for VMs build using PyPy, it is currently still in development [6]. Comparing the execution time of each opcode, a steady performance optimization is visible in Java. The different distribution of the executed opcodes in this benchmark (Table 2 on page 28) allows Java to further improve the result. Java can optimize the most frequently executed opcodes, which weigh heavier on the total result in this benchmark than in the first one.

5.3 JIT Comparison

In order to show the impact of a JIT to the results of the benchmarks, we run in this section the equally distributed opcode set of Section 5.1 on page 18 with different execution counts. As the number of execution grows, the difference between the Java and the translated Python version shrinks, and eventually the Java version gets faster than the translated version.

Executions	RPython	Java 6	Ratio
1	0.00017	0.00058	0.30
10	0.00067	0.0036	0.19
100	0.0054	0.030	0.18
1000	0.054	0.21	0.26
10'000	0.54	0.82	0.65
100'000	5.4	3.1	1.74
1'000'000	54.0	25.4	2.13
10'000'000	540	248	2.18
100'000'000	5400	2462	2.19

Table 1: Influence of the JIT to the benchmark results. Average execution in Seconds over 10 runs per test. For this benchmark the equally distributed set has been used. The last two values for RPython are estimated by linearly extending the previous values

Table 1 shows the ratio of both execution times. It is clearly visible that the translated RPython version runs linear, whereas the Java version behaves differently. Other than expected, the Java version gets hot after a rather high amount of executions. We expected around 10'000 executions, but running this benchmark with finer grained steps resulted in 300'000 executions for warming up the JIT. Although the Java version features a JIT, it is only around two times faster than the our implementation. This difference is negligible with future versions of PyPy featuring a JIT.

6 Future Work

In this section we discuss the future work needed for PyGirl and corresponding tasks for PyPy.

6.1 PyGirl

The current implementation is not fully usable yet. First of all, there is no implementation yet of the sound unit available in our emulation of the device. We chose to postpone the development of this unit since all other parts had a much higher priority. Surely this unit has to be considered as an important task for completing the system.
From all the hardware parts which already do have a software counterpart, only the video driver is not working properly yet. It is currently able to display simple graphics and supports most operations. Still there are some hidden bugs, which occur only after a certain combination of instructions. Most of the bit-operations have been replaced by more readable boolean comparisons, like described in Section 4.2.3 on page 13. The graphic part, however, still relies on pixel operations and raw memory access, much like the original version. In order to create a fully object-oriented implementation, a replacement of each sprite with a corresponding sprite-object is required. The video-chip itself also supports more abstract features like a masking window and a background which should be represented by objects too.

6.2 PyPy

The current state of the implementation only allows the code of PyGirl to be translated using the C-backend. This is due to the fact that the I/O drivers for PyPy has only been completed for that backend. The fact that we only need very low level support for most VMs, like simple bitblitting, makes enabling graphics support for the other backends a straightforward task.
Translating PyGirl with the JVM as target, would make it possible to compare the performance of the original Java implementation directly with our approach. Another important factor is the availability and wide spread distribution of the Java Virtual Machine. It would be interesting to be able to run PyGirl on a mobile device [7]. Since the Java Mobile Edition is for the biggest part a subset of the Standard Edition, the toolchain needs only mi-

nor changes. Because our implementation does not rely on features which are not available on the Mobile Edition, PyGirl should be translatable in the same way to C, Java SE and Java ME.

7 Conclusion

In this paper we have shown that the usage of high-level models for the definition of whole-system VMs will increase understandability, maintainability and extendability through separation of the implementation details and the actual platform specific model.
Our prototype model PyGirl is implemented in a high-level language without the incorporation of any standard VM optimizations. Because of the strong metaprogramming capabilities of the model definition language, we were able to abstract away clear patterns in the mapping between opcodes and their semantics.
All the standard VM implementation tricks are abstracted away from the actual model to the used translation toolkit PyPy. This toolchain reintroduces all implementation details to our VM at model transformation time, thus still resulting in a highly optimized binary. We validated the approach by comparing the performance of our VM written for PyPy with a standard implementation of a VM in Java emulating the same hardware. Our benchmarks showed that both implementations result in the same magnitude of execution performance.
We showed that the combination of a high-level executable model and a translation toolchain can serve as a new approach for creating flexible and maintainable whole-system VMs for different platforms.

References

[1] J. E. Smith and R. Nair, *Virtual Machines*. Morgan Kaufmann, 2005.

[2] C. F. Bolz, A. Kuhn, A. Lienhard, N. D. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, and T. Verwaest, "Back to the future in one week — implementing a smalltalk vm in pypy," in *Workshop on Self-sustaining Systems (S3) 2008*, p. TBA, TBA, 2008. To appear.

[3] A. Rigo and S. Pedroni, "PyPy's approach to virtual machine construction," in *Proceedings of the 2006 conference on Dynamic languages symposium, OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, (New York, NY, USA), pp. 944–953, ACM, 2006.

[4] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "Back to the future: The story of Squeak, a practical Smalltalk written in itself," in *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pp. 318–326, ACM Press, Nov. 1997.

[5] C. Hasan, "Jmario," 2007.

[6] S. Pedroni and A. Rigo, "JIT compiler architecture," tech. rep., PyPy Consortium, 2007. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.

[7] L. Aubry, D. Douard, and A. Fayolle, "Case study on using pypy for embedded devices," tech. rep., PyPy Consortium, 2007.

8 How to run PyGirl

PyGirl requires Python version 2.5 or higher, the sources are available on the codespeak svn-repository. To run the project you have to checkout the full PyPy sources

```
> svn co http://codespeak.net/svn/pypy/dist pypy-dist
> cd pypy-dist
```

To translate PyGirl you to go to the translation goal folder and then run the translation script

```
> cd pypy/translator/goal
> ./translate.py --gc=generation --batch \
   targetgbimplementation.py
```

After translating you can run the binary file with one of the test roms which are located in `pypy/lang/gameboy/rom`

```
> ./targetgbimplementation-c \
  ../../lang/gameboy/rom/rom8/rom8.gb
```

To stop the emulation hit the ESC-key.
The full project's sources are located under `pypy/lang/gameboy`

9 How to run the Test-cases

To run the test-cases make sure you checked out the latest version of the PyPy distribution, including the PyGirl project. Please make sure that `pypy-dist/py` is in your Python include path. Go to the gameboy-folder and run `py.test`

```
> cd pypy-dist/pypy/lang/gameboy
> py.test
```

10 How to run JMario

To run the original Java version, you have to download the sources from the sourceforge project website

```
http://sourceforge.net/projects/mario/
```

Or you can download the sources directly from the cvs repository

```
> cvs -z3 \
     -d:pserver:anonymous@mario.cvs.sourceforge.net:\
     /cvsroot/mario co -P jmario
```

To run the script you have to change to the jmario folder and run the ant buildfile.

```
> cd jmario/
> ant
```

Then the compiled jar is available in the build folder. Type the following commands to run the jar with a test-ROM. Simple test-ROMs are included in the PyPy distribution. Test-ROMs number 8 and 9 have graphical output, the others only serve for testing the opcodes

```
> cd build/
> java -jar jmario.jar path/to/your/testROM.gb
```

11 How to run the Benchmarks

In this section we show how to use the benchmark tools to reproduce the results from Section 5 on page 18. Since PyPy is an evolving system, the results of the newest version might be different from the ones presented in the benchmark section.
In order to run the benchmarks properly you have to download the sources first. Since the profiler depends on a certain structure, the PyPy project and the JMario are packaged in this repository. Also the JMario implementation has an extended build file and some modified sources, targeting the benchmarks.

```
> svn co https://www.iam.unibe.ch/scg/svn_repos/Students/cami/pyGirl
> ls profiling/
    ...
    profileAll.sh
    profileEach.sh
    profileJIT.sh
    profileTypical.sh
    ...
```

To run a benchmark, execute one of the listend bash scripts. Each script takes the number of test probes and the number of executions as arguments. The execution is then averaged over the number of test probes. To run 10 test probes which execute each test 100 times, type the following:

```
> profiling/profileAll.sh 10 100
-------------------------------------------------------
     java1.6
-------------------------------------------------------
BUILD SUCCESSFUL
Total time: 2 seconds
    Starting 10 test runs: ..........
    Concatenating results
```

The results are stored into files under `ARCHITECTURE/result/PROFILE_XYZ.txt`.

```
> cat profiling/java1.6/result/PROFILE_All.txt
    0.2054  0.0030
```

So this test took 0.2054 Seconds and the standard deviation over the 10 test runs is ± 0.0030 Seconds.
Each of the profiling scripts automatically creates the binaries necessary to run the benchmarks. To create the binaries manually, either run the ant script in the JMARIO project with the profiling target,

```
> cd jmario; ant dist-profile
```

or the profiling target in the PYPY project, which needs a running version of the developer version of Python installed.

```
> cd pypy-dist/pypy/translator/goal/
> ./translate.py --gc=generation targetgbprofiling
```

Further instruction on translating the sources can be found on the PYPY website under[6].

12 Performance Evaluation

The following list shows the the 30 most executed opcodes summed over four different Games. Each game was allowed to execute 500'000 operations. Table 2 shows the results of this analyzation, which we used to compare the different VM implementations (Section 5 on page 18).

[6] `http://codespeak.net/pypy/dist/pypy/doc/getting-started.html`

Name	opcode	Count
Reset Program Counter	0xff	911896
Conditional Jump	0x20	260814
Memory Read	0xf0	166327
Compare	0xfe	166263
Memory Write	0x13	74595
Increment Register	0x12	74582
Memory Read	0x2a	72546
OR	0xb1	70495
Decrement Register	0xb	70487
Register Write	0x78	70487
Decrement Register	0x5	24998
Memory Write	0x32	24962
Relative Conditional Jump	0x38	4129
Decrement Register	0xd	3170
Memory Write	0x22	1034
Call Subroutine	0xcd	308
Fetch	0x21	294
Return Subroutine	0xc9	292
Push Register	0xf5	284
Pop Register	0xf1	282
Jump	0xc3	277
Memory Write	0x77	275
Memory Read	0x7e	261
Increment Register	0x3c	260
Write Memory	0xe0	88
Register Write	0x3e	55
Write Memory	0xea	47
XOR	0xaf	45
Memory Write	0x70	40
Register Write	0x7d	40

Table 2: Typical set of the most executed opcodes, summed over four different emulations.